PRACTICAL 1

Aim:- Performing matrix multiplication and finding eigen vectors and eigen values using TensorFlow

Code:-

```
# importing numpy library
import numpy as np
# create numpy 2d-array
m = np.array([[1, 2],[2, 3]])
print("Printing the Original square array:\n",m)
print()
print('**************************************')
print()
# finding eigenvalues and eigenvectors
w, v = np.linalg.eig(m)
# printing eigen values
print("Printing the Eigen values of the given square array:\n",w)
print()
# printing eigen vectors
print("Printing Right Eigen Vectors of the given square array:\n",v)
# importing numpy library
import numpy as np
# create numpy 2d-array
m = np.array([[1, 2, 3],
        [2, 3, 4],
        [4, 5, 6]])
print("Printing the Original square array:\n",m)
print()
print('**************************************')
print()
# finding eigenvalues and eigenvectors
w, v = np.linalg.eig(m)
# printing eigen values
print("Printing the Eigen values of the given square array:\n",w)
```

```
print()
# printing eigen vectors
print("Printing Right eigenvectors of the given square array:\n",v)
import tensorflow as tf
e_matrix_A = tf.random.uniform([2, 2], minval=3, maxval=10, dtype=tf.float32,
name="matrixA")
print("Matrix A: \n{}\n\n".format(e_matrix_A))
# Calculating the eigen values and vectors using tf.linalg.eigh, if you only want the values
you can use eigvalsh
eigen_values_A, eigen_vectors_A = tf.linalg.eigh(e_matrix_A)
print("Eigen Vectors: \n{} \n\nEigen Values: \n{}\n".format(eigen_vectors_A,
eigen_values_A))
# Calculating the eigen values and vectors using tf.linalg.eigh, if you only want the values
you can use eigvalsh
eigen_values_A, eigen_vectors_A = tf.linalg.eigh(e_matrix_A)
print("Eigen Vectors: \n{} \n\nEigen Values: \n{}\n".format(eigen_vectors_A,
eigen_values_A))
```

output:-

```
Printing the Original square array:
 [[1 2]
 [2 3]]

****************************************

Printing the Eigen values of the given square array:
 [-0.23606798  4.23606798]

Printing Right Eigen Vectors of the given square array:
 [[-0.85065081 -0.52573111]
 [ 0.52573111 -0.85065081]]
Printing the Original square array:
 [[1 2 3]
 [2 3 4]
 [4 5 6]]

****************************************

Printing the Eigen values of the given square array:
 [ 1.08309519e+01 -8.30951895e-01  1.01486082e-16]

Printing Right eigenvectors of the given square array:
 [[ 0.34416959  0.72770285  0.40824829]
 [ 0.49532111  0.27580256 -0.81649658]
 [ 0.79762415 -0.62799801  0.40824829]]
Matrix A:
[[9.602425  6.19266  ]
 [5.308148  3.7283862]]


Eigen Vectors:
[[-0.5078697  0.8614339]
 [ 0.8614339  0.5078697]]
```

```
Eigen Values:
[ 0.598898 12.731914]

Matrix A:
[[4.9815197 4.651471  4.916127 ]
 [4.5983944 5.4188147 5.1733828]
 [9.358104  7.6849785 9.921461 ]]


Eigen Vectors:
[[ 0.6887056   0.5047153   0.5205259 ]
 [ 0.2575503  -0.84140515  0.4750843 ]
 [-0.6777554   0.19313161  0.70947003]]

Eigen Values:
[-2.5081651   0.89650935 21.933455   ]
```

PRACTICAL 2

Aim:- Solving XOR problem using deep feed forward network

Code:-

```
# importing Python library
import numpy as np
# define Unit Step Function
def unitStep(v):
    if v >= 0:
        return 1
    else:
        return 0
# design Perceptron Model
def perceptronModel(x, w, b):
    v = np.dot(w, x) + b
    y = unitStep(v)
    return y
# NOT Logic Function
# wNOT = -1, bNOT = 0.5
def NOT_logicFunction(x):
    wNOT = -1
    bNOT = 0.5
    return perceptronModel(x, wNOT, bNOT)
# AND Logic Function
# here w1 = wAND1 = 1,
# w2 = wAND2 = 1, bAND = -1.5
def AND_logicFunction(x):
    w = np.array([1, 1])
    bAND = -1.5
    return perceptronModel(x, w, bAND)
# OR Logic Function
# w1 = 1, w2 = 1, bOR = -0.5
def OR_logicFunction(x):
    w = np.array([1, 1])
```

```
    bOR = -0.5
    return perceptronModel(x, w, bOR)
# XOR Logic Function
# with AND, OR and NOT
# function calls in sequence
def XOR_logicFunction(x):
  y1 = AND_logicFunction(x)
  y2 = OR_logicFunction(x)
  y3 = NOT_logicFunction(y1)
  final_x = np.array([y2, y3])
  finalOutput = AND_logicFunction(final_x)
  y3 = NOT_logicFunction(y1)
  return finalOutput
# testing the Perceptron Model
test1 = np.array([0, 1])
test2 = np.array([1, 1])
test3 = np.array([0, 0])
test4 = np.array([1, 0])
print("XOR({}, {}) = {}".format(0, 1, XOR_logicFunction(test1)))
print("XOR({}, {}) = {}".format(1, 1, XOR_logicFunction(test2)))
print("XOR({}, {}) = {}".format(0, 0, XOR_logicFunction(test3)))
print("XOR({}, {}) = {}".format(1, 0, XOR_logicFunction(test4)))
```

output:-

```
XOR(0, 1) = 1
XOR(1, 1) = 0
XOR(0, 0) = 0
XOR(1, 0) = 1
```

PRACTICAL 3

Aim:- Implementing deep neural network for performing binary classification task.

Code:-

```
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = pd.read_csv('sonar.all-data', header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# baseline model
def create_baseline():
	# create model
	model = Sequential()
	model.add(Dense(60, input_dim=60, activation='relu'))
	model.add(Dense(1, activation='sigmoid'))
	# Compile model
	model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
	return model
estimator = KerasClassifier(build_fn=create_baseline, epochs=100, batch_size=5, verbose=0)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
```

```
results = cross_val_score(estimator, X, encoded_Y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_baseline, epochs=100,
batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Standardized: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
# smaller model
def create_smaller():
	# create model
	model = Sequential()
	model.add(Dense(30, input_dim=60, activation='relu'))
	model.add(Dense(1, activation='sigmoid'))
	# Compile model
	model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
	return model
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_smaller, epochs=100,
batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Smaller: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
# larger model
def create_larger():
	# create model
	model = Sequential()
	model.add(Dense(60, input_dim=60, activation='relu'))
	model.add(Dense(30, activation='relu'))
	model.add(Dense(1, activation='sigmoid'))
```

```
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_larger, epochs=100, batch_size=5,
verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Larger: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

output:-

```
Baseline: 82.64% (7.02%)

Standardized: 86.10% (5.38%)

Smaller: 87.57% (8.56%)

Larger: 83.17% (8.33%)
```

PRACTICAL 4

4a.

Aim:- Using deep feed forward network with two hidden layers for performing classification and predicting the class

Code:-

```
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import make_blobs
from sklearn.preprocessing import MinMaxScaler
X,Y=make_blobs(n_samples=100,centers=2,n_features=2,random_state=1)
scalar=MinMaxScaler()
scalar.fit(X)
X=scalar.transform(X)
model=Sequential()
model.add(Dense(4,input_dim=2,activation='relu'))
model.add(Dense(4,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam')
model.summary()
model.fit(X,Y,epochs=100)
Xnew,Yreal=make_blobs(n_samples=3,centers=2,n_features=2,random_state=1)
Xnew=scalar.transform(Xnew)
Ynew=model.predict(Xnew)
for i in range(len(Xnew)):
  print("X=%s,Predicted=%s,Desired=%s"%(Xnew[i],Ynew[i],Yreal[i]))
```

Output:-

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 4)                 12

 dense_1 (Dense)             (None, 4)                 20

 dense_2 (Dense)             (None, 1)                 5

=================================================================
Total params: 37 (148.00 Byte)
Trainable params: 37 (148.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/100
4/4 [==============================] - 1s 8ms/step - loss: 0.6746
Epoch 2/100
4/4 [==============================] - 0s 4ms/step - loss: 0.6727
Epoch 3/100
4/4 [==============================] - 0s 4ms/step - loss: 0.6706
Epoch 4/100
4/4 [==============================] - 0s 4ms/step - loss: 0.6690
Epoch 5/100
4/4 [==============================] - 0s 4ms/step - loss: 0.6670
Epoch 6/100
4/4 [==============================] - 0s 4ms/step - loss: 0.6652
Epoch 7/100
4/4 [==============================] - 0s 4ms/step - loss: 0.6635
Epoch 8/100
4/4 [==============================] - 0s 4ms/step - loss: 0.6617
Epoch 9/100
4/4 [==============================] - 0s 4ms/step - loss: 0.6598
Epoch 10/100
```

```
X=[0.89337759 0.65864154],Predicted=[0.18565549],Desired=0
X=[0.29097707 0.12978982],Predicted=[0.55316097],Desired=1
X=[0.78082614 0.75391697],Predicted=[0.17875534],Desired=0
```

4B

Aim:- Using deep feed forward network with two hidden layers for performing classification and predicting the probability of class.

Code:-

```
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import make_blobs
from sklearn.preprocessing import MinMaxScaler
X,Y=make_blobs(n_samples=100,centers=2,n_features=2,random_state=1)
scalar=MinMaxScaler()
scalar.fit(X)
X=scalar.transform(X)
model=Sequential()
model.add(Dense(4,input_dim=2,activation='relu'))
model.add(Dense(4,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam')
```

```
model.summary()
model.fit(X,Y,epochs=200)
Xnew,Yreal=make_blobs(n_samples=3,centers=2,n_features=2,random_state=1)
new=scalar.transform(Xnew)
Yclass=model.predict(Xnew)
import numpy as np
def predict_prob(number):
  return [number[0],1-number[0]]
y_prob = np.array(list(map(predict_prob, model.predict(Xnew))))
y_prob
for i in range(len(Xnew)):
print("X=%s,Predicted_probability=%s,Predicted_class=%s"%(Xnew[i],y_prob[i],Yclass[i]))
predict_prob=model.predict([Xnew])
predict_classes=np.argmax(predict_prob,axis=1)
predict_classes
```

Output:-

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 4)                 12

 dense_1 (Dense)             (None, 4)                 20

 dense_2 (Dense)             (None, 1)                 5

=================================================================
Total params: 37 (148.00 Byte)
Trainable params: 37 (148.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/200
4/4 [==============================] - 3s 14ms/step - loss: 0.7702
Epoch 2/200
4/4 [==============================] - 0s 7ms/step - loss: 0.7660
Epoch 3/200
4/4 [==============================] - 0s 8ms/step - loss: 0.7620
Epoch 4/200
4/4 [==============================] - 0s 7ms/step - loss: 0.7583
```

```
X=[-0.79415228  2.10495117],Predicted_probability=[0.05700015 0.94299985],Predicted_class=[0.05700015]
1/1 [==============================] - 0s 170ms/step
X=[-8.25290074 -4.71455545],Predicted_probability=[0.83802909 0.16197091],Predicted_class=[0.8380291]
1/1 [==============================] - 0s 72ms/step
X=[-2.18773166  3.33352125],Predicted_probability=[0.00778263 0.99221737],Predicted_class=[0.00778263]
1/1 [==============================] - 0s 41ms/step
array([0, 0, 0])
```

PRACTICAL 5

Aim:- Evaluating feed forward deep network for regression using KFold cross validation.

Code:-

```
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
batch_size = 50
img_width, img_height, img_num_channels = 32, 32, 3
loss_function = sparse_categorical_crossentropy
no_classes = 100
no_epochs = 10   # you can increase it to 20,50,70, 100
optimizer = Adam()
verbosity = 1
# Load CIFAR-10 data
(input_train, target_train), (input_test, target_test) = cifar10.load_data()
# Determine shape of the data
input_shape = (img_width, img_height, img_num_channels)
# Parse numbers as floats
input_train = input_train.astype('float32')
input_test = input_test.astype('float32')
# Normalize data
input_train = input_train / 255
input_test = input_test / 255
# Create the model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
```

```
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(no_classes, activation='softmax'))
model.summary()
# Compile the model
model.compile(loss=loss_function, optimizer=optimizer,metrics=['accuracy'])
# Fit data to model (this will take little time to train)
history = model.fit(input_train, target_train, batch_size=batch_size, epochs=no_epochs,
verbose=verbosity)
# Generate generalization metrics
score = model.evaluate(input_test, target_test, verbose=0)
print(f'Test loss: {score[0]} / Test accuracy: {score[1]}')
# Visualize history
# Plot history: Loss
plt.plot(history.history['loss'])
plt.title('Validation loss history')
plt.ylabel('Loss value')
plt.xlabel('No. epoch')
plt.show()
# Plot history: Accuracy
plt.plot(history.history['accuracy'])
plt.title('Validation accuracy history')
plt.ylabel('Accuracy value (%)')
plt.xlabel('No. epoch')
plt.show()
# By Adding k fold cross validation
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import KFold
import numpy as np
# Model configuration
```

```
batch_size = 50
img_width, img_height, img_num_channels = 32, 32, 3
loss_function = sparse_categorical_crossentropy
no_classes = 100
no_epochs = 10
optimizer = Adam()
verbosity = 1
num_folds = 5
# Load CIFAR-10 data
(input_train, target_train), (input_test, target_test) = cifar10.load_data()
# Determine shape of the data
input_shape = (img_width, img_height, img_num_channels)
# Parse numbers as floats
input_train = input_train.astype('float32')
input_test = input_test.astype('float32')
# Normalize data
input_train = input_train / 255
input_test = input_test / 255
# Define per-fold score containers
acc_per_fold = []
loss_per_fold = []
# Merge inputs and targets
inputs = np.concatenate((input_train, input_test), axis=0)
targets = np.concatenate((target_train, target_test), axis=0)
# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)
# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(inputs, targets):
  # Define the model architecture
  model = Sequential()
  model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
```

```
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Flatten())
  model.add(Dense(256, activation='relu'))
  model.add(Dense(128, activation='relu'))
  model.add(Dense(no_classes, activation='softmax'))
  # Compile the model
  model.compile(loss=loss_function,
              optimizer=optimizer,
              metrics=['accuracy'])
 # Generate a print
  print('------------------------------------------------------------------------')
  print(f'Training for fold {fold_no} ...')
  # Fit data to model
  history = model.fit(inputs[train], targets[train],
            batch_size=batch_size,
            epochs=no_epochs,
            verbose=verbosity)
  # Generate generalization metrics
  scores = model.evaluate(inputs[test], targets[test], verbose=0)
  print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]};
{model.metrics_names[1]} of {scores[1]*100}%')
  acc_per_fold.append(scores[1] * 100)
  loss_per_fold.append(scores[0])
  # Increase fold number
  fold_no = fold_no + 1
# == Provide average scores ==
print('------------------------------------------------------------------------')
print('Score per fold')
for i in range(0, len(acc_per_fold)):
  print('------------------------------------------------------------------------')
  print(f'> Fold {i+1} - Loss: {loss_per_fold[i]} - Accuracy: {acc_per_fold[i]}%')
print('------------------------------------------------------------------------')
print('Average scores for all folds:')
print(f'> Accuracy: {np.mean(acc_per_fold)} (+- {np.std(acc_per_fold)})')
```

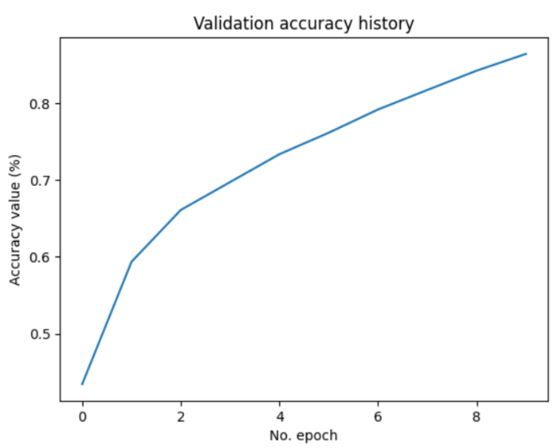
print(f'> Loss: {np.mean(loss_per_fold)}')

print('--')

Output:-

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 7s 0us/step
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2  (None, 15, 15, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 6, 6, 64)          0
 g2D)

 flatten (Flatten)           (None, 2304)              0

 dense (Dense)               (None, 256)               590080

 dense_1 (Dense)             (None, 128)               32896

 dense_2 (Dense)             (None, 100)               12900

=================================================================
Total params: 655268 (2.50 MB)
Trainable params: 655268 (2.50 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/10
1000/1000 [==============================] - 71s 67ms/step - loss: 1.5742 - accuracy: 0.4341
Epoch 2/10
1000/1000 [==============================] - 68s 68ms/step - loss: 1.1437 - accuracy: 0.5935
Epoch 3/10
```

```
_________________________________________________________________
Epoch 1/10
1000/1000 [==============================] - 71s 67ms/step - loss: 1.5742 - accuracy: 0.4341
Epoch 2/10
1000/1000 [==============================] - 68s 68ms/step - loss: 1.1437 - accuracy: 0.5935
Epoch 3/10
1000/1000 [==============================] - 66s 66ms/step - loss: 0.9719 - accuracy: 0.6611
Epoch 4/10
1000/1000 [==============================] - 67s 67ms/step - loss: 0.8626 - accuracy: 0.6975
Epoch 5/10
1000/1000 [==============================] - 66s 66ms/step - loss: 0.7591 - accuracy: 0.7338
Epoch 6/10
1000/1000 [==============================] - 66s 66ms/step - loss: 0.6754 - accuracy: 0.7618
Epoch 7/10
1000/1000 [==============================] - 68s 68ms/step - loss: 0.5929 - accuracy: 0.7921
Epoch 8/10
1000/1000 [==============================] - 66s 66ms/step - loss: 0.5166 - accuracy: 0.8175
Epoch 9/10
1000/1000 [==============================] - 66s 66ms/step - loss: 0.4470 - accuracy: 0.8427
Epoch 10/10
1000/1000 [==============================] - 66s 66ms/step - loss: 0.3804 - accuracy: 0.8645
Test loss: 1.0700150728225708 / Test accuracy: 0.6970999836921692
```



```
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 1.0987507104873657 - Accuracy: 68.94999742507935%
------------------------------------------------------------------------
> Fold 2 - Loss: 1.0545800924301147 - Accuracy: 71.17499709129333%
------------------------------------------------------------------------
> Fold 3 - Loss: 1.082228660583496 - Accuracy: 69.87500190734863%
------------------------------------------------------------------------
> Fold 4 - Loss: 1.1083070039749146 - Accuracy: 69.70833539962769%
------------------------------------------------------------------------
> Fold 5 - Loss: 1.1079905033111572 - Accuracy: 70.35833597183228%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 70.01333355903625 (+- 0.7363874094375512)
> Loss: 1.0903713941574096
------------------------------------------------------------------------
```

PRACTICAL 6

Aim:- Implementing regularization to avoid overfitting in binary classification using TensorFlow.

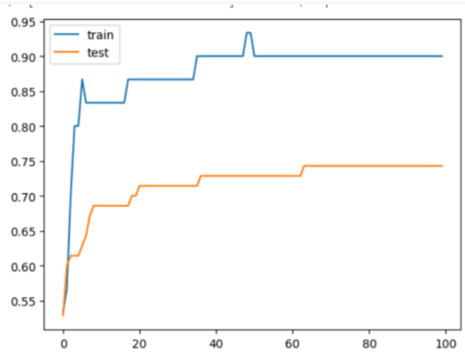
Code:-

```
from matplotlib import pyplot
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
X,Y=make_moons(n_samples=100,noise=0.2,random_state=1)
n_train=30
trainX,testX=X[:n_train,:],X[n_train:]
trainY,testY=Y[:n_train],Y[n_train:]
print(trainX.shape)
print(trainY.shape)
print(testX.shape)
print(testY.shape)
model=Sequential()
model.add(Dense(500,input_dim=2,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=100)
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
from keras.regularizers import l2
model=Sequential()
model.add(Dense(500,input_dim=2,activation='relu',kernel_regularizer=l2(0.001)))
model.add(Dense(1,activation='sigmoid'))
model.summary()
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=100)
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
```

```
pyplot.legend()
pyplot.show()
from keras.regularizers import l1_l2
model=Sequential()
model.add(Dense(500,input_dim=2,activation='relu',kernel_regularizer=l1_l2(l1=0.001,l2=0.
001)))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
model.summary()
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=100)
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
```

Output:-

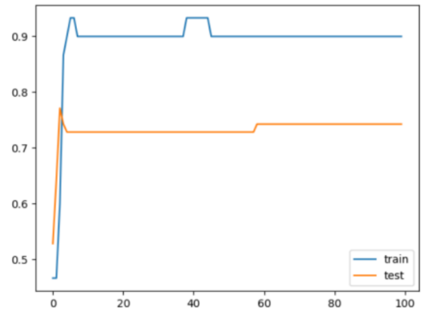
```
(30, 2)
(30,)
(70, 2)
(70,)
Epoch 1/100
1/1 [==============================] - 1s 1s/step - loss: 0.6976 - accuracy: 0.5333 - val_loss: 0.6923 - val_accuracy: 0.5286
Epoch 2/100
1/1 [==============================] - 0s 63ms/step - loss: 0.6813 - accuracy: 0.5667 - val_loss: 0.6818 - val_accuracy: 0.6000
Epoch 3/100
1/1 [==============================] - 0s 75ms/step - loss: 0.6654 - accuracy: 0.7000 - val_loss: 0.6717 - val_accuracy: 0.6143
Epoch 4/100
1/1 [==============================] - 0s 60ms/step - loss: 0.6500 - accuracy: 0.8000 - val_loss: 0.6619 - val_accuracy: 0.6143
Epoch 5/100
1/1 [==============================] - 0s 64ms/step - loss: 0.6349 - accuracy: 0.8000 - val_loss: 0.6524 - val_accuracy: 0.6143
Epoch 6/100
1/1 [==============================] - 0s 72ms/step - loss: 0.6203 - accuracy: 0.8667 - val_loss: 0.6433 - val_accuracy: 0.6286
Epoch 7/100
1/1 [==============================] - 0s 47ms/step - loss: 0.6061 - accuracy: 0.8333 - val_loss: 0.6345 - val_accuracy: 0.6429
Epoch 8/100
1/1 [==============================] - 0s 48ms/step - loss: 0.5922 - accuracy: 0.8333 - val_loss: 0.6260 - val_accuracy: 0.6714
Epoch 9/100
1/1 [==============================] - 0s 60ms/step - loss: 0.5787 - accuracy: 0.8333 - val_loss: 0.6178 - val_accuracy: 0.6857
Epoch 10/100
1/1 [==============================] - 0s 48ms/step - loss: 0.5656 - accuracy: 0.8333 - val_loss: 0.6099 - val_accuracy: 0.6857
Epoch 11/100
```




```
Model: "sequential_1"
```

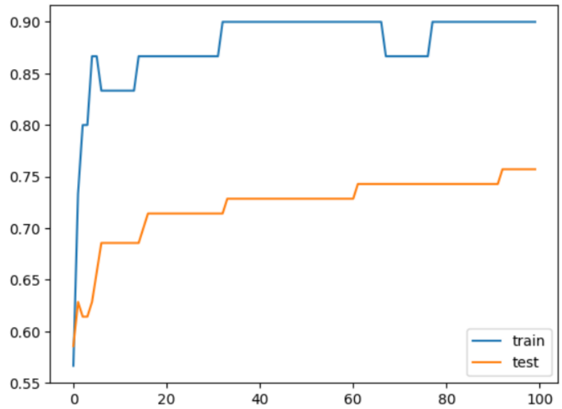
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_2 (Dense)             (None, 500)               1500

 dense_3 (Dense)             (None, 1)                 501

=================================================================
Total params: 2001 (7.82 KB)
Trainable params: 2001 (7.82 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/100
1/1 [==============================] - 1s 1s/step - loss: 0.7094 - accuracy: 0.4667 - val_loss: 0.6920 - val_accuracy: 0.5286
Epoch 2/100
1/1 [==============================] - 0s 62ms/step - loss: 0.6922 - accuracy: 0.4667 - val_loss: 0.6808 - val_accuracy: 0.6429
Epoch 3/100
1/1 [==============================] - 0s 64ms/step - loss: 0.6756 - accuracy: 0.6000 - val_loss: 0.6700 - val_accuracy: 0.7714
Epoch 4/100
1/1 [==============================] - 0s 56ms/step - loss: 0.6595 - accuracy: 0.8667 - val_loss: 0.6595 - val_accuracy: 0.7429
Epoch 5/100
1/1 [==============================] - 0s 114ms/step - loss: 0.6438 - accuracy: 0.9000 - val_loss: 0.6495 - val_accuracy: 0.7286
```

```
Epoch 99/100
1/1 [==============================] - 0s 54ms/step - loss: 0.1971 - accuracy: 0.9000 - val_loss: 0.4361 - val_accuracy: 0.7429
Epoch 100/100
1/1 [==============================] - 0s 53ms/step - loss: 0.1964 - accuracy: 0.9000 - val_loss: 0.4356 - val_accuracy: 0.7429
```




```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_4 (Dense)             (None, 500)               1500

 dense_5 (Dense)             (None, 1)                 501

=================================================================
Total params: 2001 (7.82 KB)
Trainable params: 2001 (7.82 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/100
1/1 [==============================] - 1s 822ms/step - loss: 0.7412 - accuracy: 0.5667 - val_loss: 0.7404 - val_accuracy: 0.5857
Epoch 2/100
1/1 [==============================] - 0s 54ms/step - loss: 0.7249 - accuracy: 0.7333 - val_loss: 0.7299 - val_accuracy: 0.6286
Epoch 3/100
1/1 [==============================] - 0s 36ms/step - loss: 0.7090 - accuracy: 0.8000 - val_loss: 0.7199 - val_accuracy: 0.6143
Epoch 4/100
1/1 [==============================] - 0s 51ms/step - loss: 0.6936 - accuracy: 0.8000 - val_loss: 0.7102 - val_accuracy: 0.6143
Epoch 5/100
1/1 [==============================] - 0s 39ms/step - loss: 0.6786 - accuracy: 0.8667 - val_loss: 0.7009 - val_accuracy: 0.6286
```

```
Epoch 99/100
1/1 [==============================] - 0s 41ms/step - loss: 0.2558 - accuracy: 0.9000 - val_loss: 0.4725 - val_accuracy: 0.7571
Epoch 100/100
1/1 [==============================] - 0s 54ms/step - loss: 0.2550 - accuracy: 0.9000 - val_loss: 0.4715 - val_accuracy: 0.7571
```



PRACTICAL 7

Aim:- Implementing Text classification with an RNN

Code:-

```
import numpy as np
import tensorflow_datasets as tfds
import tensorflow as tf
tfds.disable_progress_bar()
import matplotlib.pyplot as plt
def plot_graphs(history, metric):
  plt.plot(history.history[metric])
  plt.plot(history.history['val_'+metric], '')
  plt.xlabel("Epochs")
  plt.ylabel(metric)
  plt.legend([metric, 'val_'+metric])
dataset, info = tfds.load('imdb_reviews', with_info=True,
                          as_supervised=True)
train_dataset, test_dataset = dataset['train'], dataset['test']
train_dataset.element_spec
for example, label in train_dataset.take(5):
  print('text: ', example.numpy())
  print('label: ', label.numpy())
BUFFER_SIZE = 10000
BATCH_SIZE = 64
for example, label in train_dataset.take(1):
  print('texts: ', example.numpy()[:3])
  print()
  print('labels: ', label.numpy()[:3])
```

```
VOCAB_SIZE = 1000
encoder = tf.keras.layers.TextVectorization(max_tokens=VOCAB_SIZE)
encoder.adapt(train_dataset.map(lambda text, label: text))
vocab = np.array(encoder.get_vocabulary())
vocab[:20]
encoded_example = encoder(example)[:3].numpy()
encoded_example
for n in range(3):
  print("Original: ", example[n].numpy())
  print("Round-trip: ", " ".join(vocab[encoded_example[n]]))
  print()
model = tf.keras.Sequential([
    encoder,
    tf.keras.layers.Embedding(
        input_dim=len(encoder.get_vocabulary()),
        output_dim=64,
        # Use masking to handle the variable sequence lengths
        mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
])
print([layer.supports_masking for layer in model.layers])
# predict on a sample text without padding.
sample_text = ('The movie was cool. The animation and the graphics '
               'were out of this world. I would recommend this movie.')
predictions = model.predict(np.array([sample_text]))
print(predictions[0])
# predict on a sample text with padding
padding = "the " * 2000
predictions = model.predict(np.array([sample_text, padding]))
print(predictions[0])
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(1e-4),
```

```
            metrics=['accuracy'])
history = model.fit(train_dataset, epochs=10,
                    validation_data=test_dataset,
                    validation_steps=30)
test_loss, test_acc = model.evaluate(test_dataset)

print('Test Loss:', test_loss)
print('Test Accuracy:', test_acc)
plt.figure(figsize=(16, 8))
plt.subplot(1, 2, 1)
plot_graphs(history, 'accuracy')
plt.ylim(None, 1)
plt.subplot(1, 2, 2)
plot_graphs(history, 'loss')
plt.ylim(0, None)
sample_text = ('The movie was cool. The animation and the graphics '
               'were out of this world. I would recommend this movie.')
predictions = model.predict(np.array([sample_text]))
model = tf.keras.Sequential([
    encoder,
    tf.keras.layers.Embedding(len(encoder.get_vocabulary()), 64, mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,  return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1)
])
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
            optimizer=tf.keras.optimizers.Adam(1e-4),
            metrics=['accuracy'])
history = model.fit(train_dataset, epochs=10,
                    validation_data=test_dataset,
                    validation_steps=30)
test_loss, test_acc = model.evaluate(test_dataset)
```

```
print('Test Loss:', test_loss)
print('Test Accuracy:', test_acc)
# predict on a sample text without padding.

sample_text = ('The movie was not good. The animation and the graphics '
               'were terrible. I would not recommend this movie.')
predictions = model.predict(np.array([sample_text]))
print(predictions)
plt.figure(figsize=(16, 6))
plt.subplot(1, 2, 1)
plot_graphs(history, 'accuracy')
plt.subplot(1, 2, 2)
plot_graphs(history, 'loss')
```

Output:

```
(TensorSpec(shape=(), dtype=tf.string, name=None),
 TensorSpec(shape=(), dtype=tf.int64, name=None))
```

```
text:  b"This was an absolutely terrible movie. Don't be lured in by Christopher Walken or Michael Ironside. Both are great actors, but this must simply be their worst role in history.
label:  0
text:  b'I have been known to fall asleep during films, but this is usually due to a combination of things including, really tired, being warm and comfortable on the sette and having
label:  0
text:  b'Mann photographs the Alberta Rocky Mountains in a superb fashion, and Jimmy Stewart and Walter Brennan give enjoyable performances as they always seem to do. <br /><br />But
label:  0
text:  b'This is the kind of film for a snowy Sunday afternoon when the rest of the world can go ahead with its own business as you descend into a big arm-chair and mellow for a couple
label:  1
text:  b'As others have mentioned, all the women that go nude in this film are mostly absolutely gorgeous. The plot very ably shows the hypocrisy of the female libido. When men are ar
label:  1
```

```
texts:  [b'Two years ago I watched "The Matador" in cinema and I loved everything about this movie. Obviously, I was totally under impression of Pier
 b"ZP is deeply related to that youth dream represented by the hippie movement.The college debate in the beginning of the movie states the cultural s
 b"As an animated film from 1978, this is pretty good--generally well above the standard of the days when Disney hadn't done anything good in years (

labels:  [1 1 1]
```

```
array(['', '[UNK]', 'the', 'and', 'a', 'of', 'to', 'is', 'in', 'it', 'i',
       'this', 'that', 'br', 'was', 'as', 'for', 'with', 'movie', 'but'],
      dtype='<U14')
```

```
array([[105, 148, 598, ...,   0,   0,   0],
       [  1,   7,   1, ...,   0,   0,   0],
       [ 15,  34,   1, ...,   0,   0,   0]])
```

```
Original:  b'Two years ago I watched "The Matador" in cinema and I loved everything about this movie. Obviously, I was totally under impression
Round-trip:  two years ago i watched the [UNK] in cinema and i loved everything about this movie obviously i was totally under [UNK] of [UNK] [U

Original:  b"ZP is deeply related to that youth dream represented by the hippie movement.The college debate in the beginning of the movie states
Round-trip:  [UNK] is [UNK] [UNK] to that [UNK] dream [UNK] by the [UNK] [UNK] [UNK] [UNK] in the beginning of the movie [UNK] the [UNK] situati

Original:  b"As an animated film from 1978, this is pretty good--generally well above the standard of the days when Disney hadn't done anything
Round-trip:  as an [UNK] film from [UNK] this is pretty [UNK] well above the [UNK] of the days when disney [UNK] done anything good in years and
```

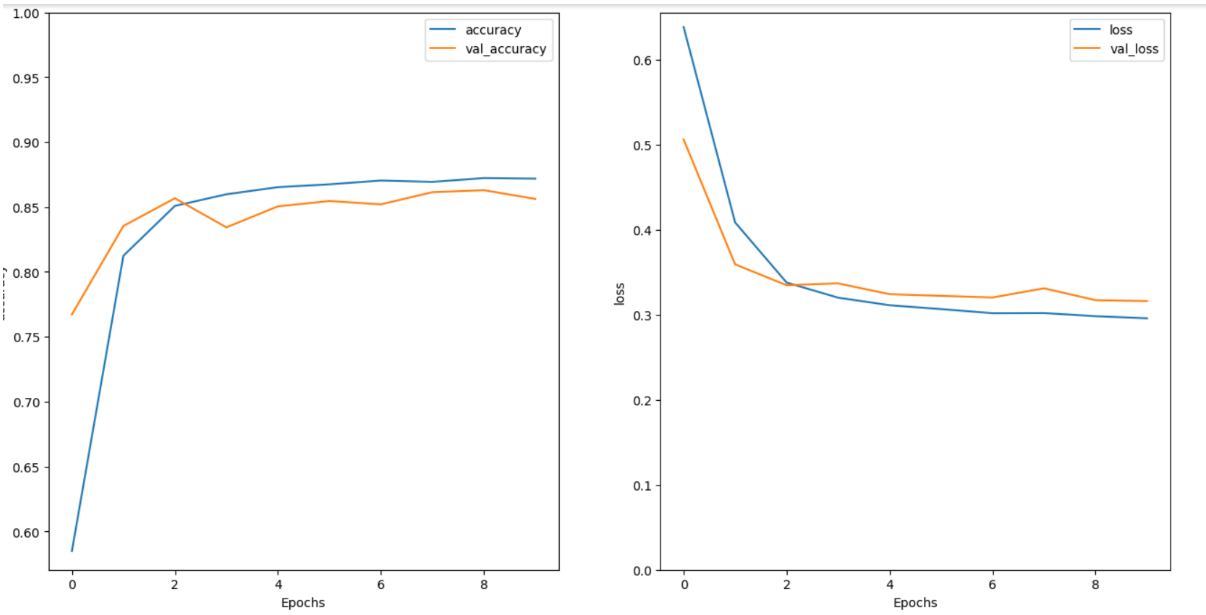
```
[False, True, True, True, True]
```

```
1/1 [==============================] - 4s 4s/step
[-0.00414192]
```

```
1/1 [==============================] - 0s 92ms/step
[-0.00414192]
```

```
Epoch 1/10
391/391 [==============================] - 52s 107ms/step - loss: 0.6382 - accuracy: 0.5848 - val_loss: 0.5060 - val_accuracy: 0.7672
Epoch 2/10
391/391 [==============================] - 29s 73ms/step - loss: 0.4085 - accuracy: 0.8124 - val_loss: 0.3592 - val_accuracy: 0.8354
Epoch 3/10
391/391 [==============================] - 28s 71ms/step - loss: 0.3378 - accuracy: 0.8509 - val_loss: 0.3346 - val_accuracy: 0.8568
Epoch 4/10
391/391 [==============================] - 30s 75ms/step - loss: 0.3200 - accuracy: 0.8598 - val_loss: 0.3368 - val_accuracy: 0.8344
Epoch 5/10
391/391 [==============================] - 27s 69ms/step - loss: 0.3111 - accuracy: 0.8653 - val_loss: 0.3241 - val_accuracy: 0.8505
Epoch 6/10
391/391 [==============================] - 27s 69ms/step - loss: 0.3065 - accuracy: 0.8675 - val_loss: 0.3221 - val_accuracy: 0.8547
Epoch 7/10
391/391 [==============================] - 27s 67ms/step - loss: 0.3017 - accuracy: 0.8704 - val_loss: 0.3202 - val_accuracy: 0.8521
Epoch 8/10
391/391 [==============================] - 26s 66ms/step - loss: 0.3019 - accuracy: 0.8694 - val_loss: 0.3310 - val_accuracy: 0.8615
Epoch 9/10
391/391 [==============================] - 27s 67ms/step - loss: 0.2982 - accuracy: 0.8723 - val_loss: 0.3171 - val_accuracy: 0.8630
Epoch 10/10
391/391 [==============================] - 26s 66ms/step - loss: 0.2957 - accuracy: 0.8719 - val_loss: 0.3160 - val_accuracy: 0.8562
```

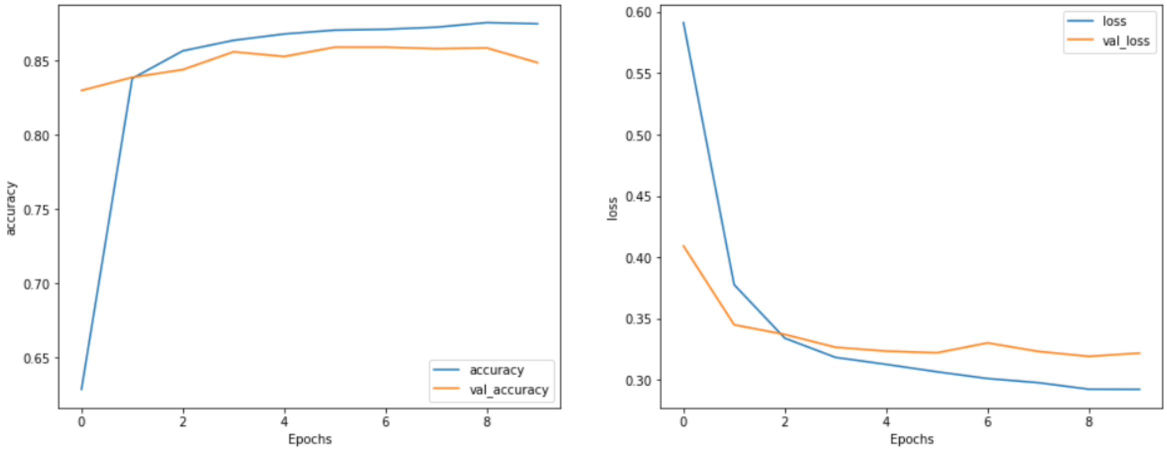
```
391/391 [==============================] - 13s 32ms/step - loss: 0.3142 - accuracy: 0.8554
Test Loss: 0.31423383951187134
Test Accuracy: 0.855400025844574
```



```
Epoch 1/10
391/391 [==============================] - 79s 158ms/step - loss: 0.6048 - accuracy: 0.6056 - val_loss: 0.4203 - val_accuracy: 0.8130
Epoch 2/10
391/391 [==============================] - 51s 130ms/step - loss: 0.3736 - accuracy: 0.8338 - val_loss: 0.3612 - val_accuracy: 0.8422
Epoch 3/10
391/391 [==============================] - 52s 133ms/step - loss: 0.3314 - accuracy: 0.8580 - val_loss: 0.3379 - val_accuracy: 0.8417
Epoch 4/10
391/391 [==============================] - 51s 130ms/step - loss: 0.3192 - accuracy: 0.8621 - val_loss: 0.3320 - val_accuracy: 0.8573
Epoch 5/10
391/391 [==============================] - 52s 132ms/step - loss: 0.3118 - accuracy: 0.8662 - val_loss: 0.3284 - val_accuracy: 0.8599
Epoch 6/10
391/391 [==============================] - 51s 129ms/step - loss: 0.3061 - accuracy: 0.8682 - val_loss: 0.3301 - val_accuracy: 0.8635
Epoch 7/10
391/391 [==============================] - 50s 127ms/step - loss: 0.3028 - accuracy: 0.8690 - val_loss: 0.3184 - val_accuracy: 0.8599
Epoch 8/10
391/391 [==============================] - 50s 129ms/step - loss: 0.2978 - accuracy: 0.8710 - val_loss: 0.3183 - val_accuracy: 0.8599
Epoch 9/10
391/391 [==============================] - 50s 128ms/step - loss: 0.2975 - accuracy: 0.8732 - val_loss: 0.3170 - val_accuracy: 0.8604
Epoch 10/10
391/391 [==============================] - 50s 127ms/step - loss: 0.2952 - accuracy: 0.8723 - val_loss: 0.3174 - val_accuracy: 0.8641
```

```
391/391 [==============================] - 22s 55ms/step - loss: 0.3138 - accuracy: 0.8614
Test Loss: 0.3138425052165985
Test Accuracy: 0.8613600134849548
```

```
1/1 [==============================] - 5s 5s/step
[[-1.5253704]]
```



PRACTICAL 8

Aim:- Implementation of Autoencoders

Code:-

```
import keras
from keras import layers
# This is the size of our encoded representations
encoding_dim = 32  # 32 floats -> compression of factor 24.5, assuming the input is 784
floats
# This is our input image
input_img = keras.Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = layers.Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = layers.Dense(784, activation='sigmoid')(encoded)
# This model maps an input to its reconstruction
autoencoder = keras.Model(input_img, decoded)
#Let's also create a separate encoder model:
# This model maps an input to its encoded representation
encoder = keras.Model(input_img, encoded)
# This is our encoded (32-dimensional) input
encoded_input = keras.Input(shape=(encoding_dim,))
```

# Retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# Create the decoder model
decoder = keras.Model(encoded_input, decoder_layer(encoded_input))
#Now let's train our autoencoder to reconstruct MNIST digits.
#First, we'll configure our model to use a per-pixel binary crossentropy loss, and the Adam optimizer:
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
#Let's prepare our input data. We're using MNIST digits, and we're discarding the labels (since we're only interested in encoding/decoding the input images).

```
from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()
# We will normalize all values between 0 and 1 and we will flatten the 28x28 images into vectors of size 784.
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
# Now let's train our autoencoder for 50 epochs:
autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
# Encode and decode some digits
# Note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
# Use Matplotlib
import matplotlib.pyplot as plt
```

```
n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
from keras import regularizers
encoding_dim = 32
input_img = keras.Input(shape=(784,))
# Add a Dense layer with a L1 activity regularizer
encoded = layers.Dense(encoding_dim, activation='relu',
                activity_regularizer=regularizers.l1(10e-5))(input_img)
decoded = layers.Dense(784, activation='sigmoid')(encoded)
autoencoder = keras.Model(input_img, decoded)
#Let's also create a separate encoder model:
# This model maps an input to its encoded representation
encoder = keras.Model(input_img, encoded)
# This is our encoded (32-dimensional) input
encoded_input = keras.Input(shape=(encoding_dim,))
# Retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# Create the decoder model
```

```
decoder = keras.Model(encoded_input, decoder_layer(encoded_input))
#Now let's train our autoencoder to reconstruct MNIST digits.
#First, we'll configure our model to use a per-pixel binary crossentropy loss, and the Adam
optimizer:
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
#Let's prepare our input data. We're using MNIST digits, and we're discarding the labels
(since we're only interested in encoding/decoding the input images).

from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()
# We will normalize all values between 0 and 1 and we will flatten the 28x28 images into
vectors of size 784.
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
# Now let's train our autoencoder for 50 epochs:
autoencoder.fit(x_train, x_train,
        epochs=50,
        batch_size=256,
        shuffle=True,
        validation_data=(x_test, x_test))
# Now let's train our autoencoder for 50 epochs:
autoencoder.fit(x_train, x_train,
        epochs=50,
        batch_size=256,
        shuffle=True,
        validation_data=(x_test, x_test))
# Encode and decode some digits
# Note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test)
```

```
decoded_imgs = decoder.predict(encoded_imgs)
# Use Matplotlib
import matplotlib.pyplot as plt

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
input_img = keras.Input(shape=(784,))
encoded = layers.Dense(128, activation='relu')(input_img)
encoded = layers.Dense(64, activation='relu')(encoded)
encoded = layers.Dense(32, activation='relu')(encoded)
decoded = layers.Dense(64, activation='relu')(encoded)
decoded = layers.Dense(128, activation='relu')(decoded)
decoded = layers.Dense(784, activation='sigmoid')(decoded)
autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train,
                epochs=100,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

```
# Encode and decode some digits
# Note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
# Use Matplotlib
import matplotlib.pyplot as plt

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

OUTPUT:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step

(60000, 784)
(10000, 784)
```

```
Epoch 1/50
235/235 [==============================] - 3s 11ms/step - loss: 0.2755 - val_loss: 0.1890
Epoch 2/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1702 - val_loss: 0.1526
Epoch 3/50
235/235 [==============================] - 4s 16ms/step - loss: 0.1439 - val_loss: 0.1334
Epoch 4/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1284 - val_loss: 0.1210
Epoch 5/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1181 - val_loss: 0.1128
Epoch 6/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1113 - val_loss: 0.1074
Epoch 7/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1066 - val_loss: 0.1035
Epoch 8/50
235/235 [==============================] - 4s 15ms/step - loss: 0.1031 - val_loss: 0.1005
Epoch 9/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1002 - val_loss: 0.0978
Epoch 10/50
235/235 [==============================] - 2s 10ms/step - loss: 0.0981 - val_loss: 0.0960
Epoch 11/50
235/235 [==============================] - 2s 10ms/step - loss: 0.0966 - val_loss: 0.0948
Epoch 12/50
235/235 [==============================] - 2s 10ms/step - loss: 0.0956 - val_loss: 0.0940
Epoch 13/50
235/235 [==============================] - 3s 15ms/step - loss: 0.0950 - val_loss: 0.0936
Epoch 14/50
235/235 [==============================] - 2s 10ms/step - loss: 0.0945 - val_loss: 0.0930
Epoch 15/50
235/235 [==============================] - 2s 10ms/step - loss: 0.0942 - val_loss: 0.0928
Epoch 16/50
235/235 [==============================] - 2s 10ms/step - loss: 0.0939 - val_loss: 0.0926
```

```
313/313 [==============================] - 1s 2ms/step
313/313 [==============================] - 0s 1ms/step
```

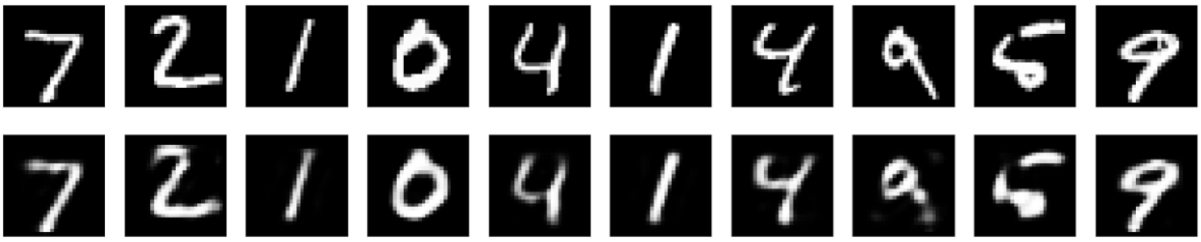
```
(60000, 784)
(10000, 784)
```

```
Epoch 1/50
235/235 [==============================] - 4s 14ms/step - loss: 0.2851 - val_loss: 0.1958
Epoch 2/50
235/235 [==============================] - 3s 13ms/step - loss: 0.1768 - val_loss: 0.1600
Epoch 3/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1519 - val_loss: 0.1421
Epoch 4/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1375 - val_loss: 0.1307
Epoch 5/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1279 - val_loss: 0.1228
Epoch 6/50
235/235 [==============================] - 3s 11ms/step - loss: 0.1214 - val_loss: 0.1174
Epoch 7/50
235/235 [==============================] - 3s 14ms/step - loss: 0.1167 - val_loss: 0.1134
Epoch 8/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1131 - val_loss: 0.1105
Epoch 9/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1106 - val_loss: 0.1083
Epoch 10/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1089 - val_loss: 0.1070
Epoch 11/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1077 - val_loss: 0.1059
Epoch 12/50
235/235 [==============================] - 3s 15ms/step - loss: 0.1068 - val_loss: 0.1051
Epoch 13/50
235/235 [==============================] - 3s 11ms/step - loss: 0.1061 - val_loss: 0.1045
Epoch 14/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1055 - val_loss: 0.1040
Epoch 15/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1050 - val_loss: 0.1036
Epoch 16/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1046 - val_loss: 0.1032
Epoch 17/50
```

```
313/313 [==============================] - 0s 1ms/step
313/313 [==============================] - 1s 2ms/step
```

```
Epoch 1/100
235/235 [==============================] - 6s 20ms/step - loss: 0.2475 - val_loss: 0.1683
Epoch 2/100
235/235 [==============================] - 4s 16ms/step - loss: 0.1492 - val_loss: 0.1353
Epoch 3/100
235/235 [==============================] - 4s 16ms/step - loss: 0.1307 - val_loss: 0.1240
Epoch 4/100
235/235 [==============================] - 5s 22ms/step - loss: 0.1222 - val_loss: 0.1173
Epoch 5/100
235/235 [==============================] - 4s 16ms/step - loss: 0.1163 - val_loss: 0.1120
Epoch 6/100
235/235 [==============================] - 4s 16ms/step - loss: 0.1122 - val_loss: 0.1091
Epoch 7/100
235/235 [==============================] - 5s 22ms/step - loss: 0.1091 - val_loss: 0.1065
Epoch 8/100
235/235 [==============================] - 4s 16ms/step - loss: 0.1064 - val_loss: 0.1040
Epoch 9/100
235/235 [==============================] - 4s 16ms/step - loss: 0.1043 - val_loss: 0.1022
Epoch 10/100
235/235 [==============================] - 5s 19ms/step - loss: 0.1026 - val_loss: 0.1004
Epoch 11/100
235/235 [==============================] - 4s 18ms/step - loss: 0.1011 - val_loss: 0.0995
Epoch 12/100
235/235 [==============================] - 4s 16ms/step - loss: 0.0997 - val_loss: 0.0980
Epoch 13/100
235/235 [==============================] - 4s 17ms/step - loss: 0.0985 - val_loss: 0.0969
Epoch 14/100
235/235 [==============================] - 5s 21ms/step - loss: 0.0973 - val_loss: 0.0964
Epoch 15/100
235/235 [==============================] - 4s 16ms/step - loss: 0.0963 - val_loss: 0.0947
Epoch 16/100
```

```
313/313 [==============================] - 0s 1ms/step
313/313 [==============================] - 0s 1ms/step
```



PRACTICAL 9

Aim:- Implementation of convolutional neural network to predict numbers from number images.

Code:-

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train.shape
y_train.shape
X_test.shape
y_test.shape
```

```
import matplotlib.pyplot as plt
plt.imshow(X_train[2])
plt.show()
plt.imshow(X_train[2], cmap=plt.cm.binary)
X_train[2]
X_train = tf.keras.utils.normalize(X_train, axis=1)
X_test = tf.keras.utils.normalize(X_test, axis=1)
plt.imshow(X_train[2], cmap=plt.cm.binary)
print(X_train[2])
import tensorflow as tf
import tensorflow.keras.layers  as KL
import tensorflow.keras.models  as KM
## Model
inputs = KL.Input(shape=(28, 28, 1))
c = KL.Conv2D(32, (3, 3), padding="valid", activation=tf.nn.relu)(inputs)
m = KL.MaxPool2D((2, 2), (2, 2))(c)
d = KL.Dropout(0.5)(m)
c = KL.Conv2D(64, (3, 3), padding="valid", activation=tf.nn.relu)(d)
m = KL.MaxPool2D((2, 2), (2, 2))(c)
d = KL.Dropout(0.5)(m)
c = KL.Conv2D(128, (3, 3), padding="valid", activation=tf.nn.relu)(d)
f = KL.Flatten()(c)
outputs = KL.Dense(10, activation=tf.nn.softmax)(f)
model = KM.Model(inputs, outputs)
model.summary()
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy",
metrics=["accuracy"])
model.fit(X_train, y_train, epochs=5)
test_loss, test_acc = model.evaluate(X_test, y_test)
print("Test Loss: {0} - Test Acc: {1}".format(test_loss, test_acc))
```

OUTPUT:

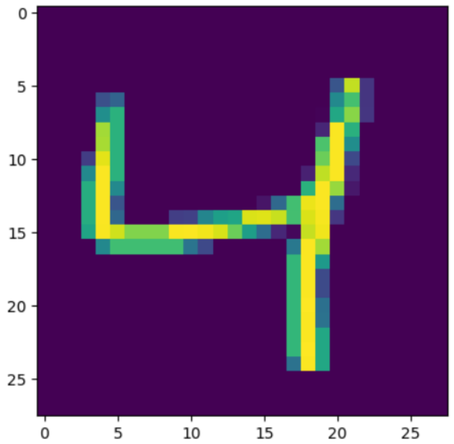
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

(60000, 28, 28)

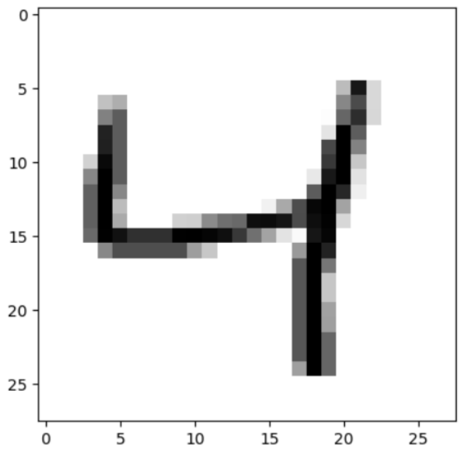
(60000,)

(10000, 28, 28)

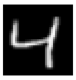
(10000,)



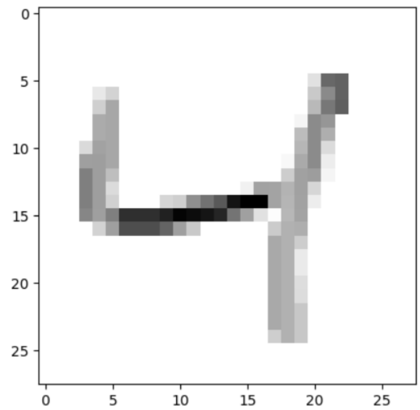
<matplotlib.image.AxesImage at 0x7f6c4b9f58d0>



ndarray (28, 28) show data



<matplotlib.image.AxesImage at 0x7f6c49892b60>



```
[[0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2  (None, 13, 13, 32)        0
 D)

 dropout (Dropout)           (None, 13, 13, 32)        0

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 64)          0
 g2D)

 dropout_1 (Dropout)         (None, 5, 5, 64)          0

 conv2d_2 (Conv2D)           (None, 3, 3, 128)         73856

 flatten (Flatten)           (None, 1152)              0

 dense (Dense)               (None, 10)                11530

=================================================================
Total params: 104202 (407.04 KB)
Trainable params: 104202 (407.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
Epoch 1/5
1875/1875 [==============================] - 64s 33ms/step - loss: 0.2601 - accuracy: 0.9169
Epoch 2/5
1875/1875 [==============================] - 64s 34ms/step - loss: 0.0992 - accuracy: 0.9684
Epoch 3/5
1875/1875 [==============================] - 62s 33ms/step - loss: 0.0756 - accuracy: 0.9764
Epoch 4/5
1875/1875 [==============================] - 63s 33ms/step - loss: 0.0648 - accuracy: 0.9796
Epoch 5/5
1875/1875 [==============================] - 61s 33ms/step - loss: 0.0569 - accuracy: 0.9814
313/313 [==============================] - 3s 9ms/step - loss: 0.0286 - accuracy: 0.9908
Test Loss: 0.028633011505007744 - Test Acc: 0.9908000230789185
```

PRACTICAL 10

Aim:- Implementing Denoising of images using Autoencoder

Code:-

```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
from __future__ import print_function
from keras.models import Model
from keras.layers import Dense, Input
from keras.datasets import mnist
from keras.regularizers import l1
from keras.optimizers import Adam
```

Utility Functions

```
def plot_autoencoder_outputs(autoencoder, n, dims):
    decoded_imgs = autoencoder.predict(x_test)
    # number of example digits to show
    n = 5
    plt.figure(figsize=(10, 4.5))
    for i in range(n):
        # plot original image
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(x_test[i].reshape(*dims))
```

```
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        if i == n/2:
            ax.set_title('Original Images')

        # plot reconstruction
        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(decoded_imgs[i].reshape(*dims))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        if i == n/2:
            ax.set_title('Reconstructed Images')
    plt.show()
def plot_loss(history):
    historydf = pd.DataFrame(history.history, index=history.epoch)
    plt.figure(figsize=(8, 6))
    historydf.plot(ylim=(0, historydf.values.max()))
    plt.title('Loss: %.3f' % history.history['loss'][-1])
def plot_compare_histories(history_list, name_list, plot_accuracy=True):
    dflist = []
    min_epoch = len(history_list[0].epoch)
    losses = []
    for history in history_list:
```

```
        h = {key: val for key, val in history.history.items() if not key.startswith('val_')}
        dflist.append(pd.DataFrame(h, index=history.epoch))
        min_epoch = min(min_epoch, len(history.epoch))
        losses.append(h['loss'][-1])
    historydf = pd.concat(dflist, axis=1)
    metrics = dflist[0].columns
    idx = pd.MultiIndex.from_product([name_list, metrics], names=['model', 'metric'])
    historydf.columns = idx
    plt.figure(figsize=(6, 8))
    ax = plt.subplot(211)
    historydf.xs('loss', axis=1, level='metric').plot(ylim=(0,1), ax=ax)
    plt.title("Training Loss: " + ' vs '.join([str(round(x, 3)) for x in losses]))
    if plot_accuracy:
        ax = plt.subplot(212)
        historydf.xs('acc', axis=1, level='metric').plot(ylim=(0,1), ax=ax)
        plt.title("Accuracy")
        plt.xlabel("Epochs")
    plt.xlim(0, min_epoch-1)
    plt.tight_layout()
```

Deep Autoencoder

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
```

```
print(x_train.shape)

print(x_test.shape)

input_size = 784

hidden_size = 128

code_size = 32

input_img = Input(shape=(input_size,))

hidden_1 = Dense(hidden_size, activation='relu')(input_img)

code = Dense(code_size, activation='relu')(hidden_1)

hidden_2 = Dense(hidden_size, activation='relu')(code)

output_img = Dense(input_size, activation='sigmoid')(hidden_2)

autoencoder = Model(input_img, output_img)

autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train, epochs=3)

plot_autoencoder_outputs(autoencoder, 5, (28, 28))

weights = autoencoder.get_weights()[0].T

n = 10

plt.figure(figsize=(20, 5))

for i in range(n):

    ax = plt.subplot(1, n, i + 1)

    plt.imshow(weights[i+0].reshape(28, 28))

    ax.get_xaxis().set_visible(False)

    ax.get_yaxis().set_visible(False)

input_size = 784

code_size = 32

input_img = Input(shape=(input_size,))
```

```
code = Dense(code_size, activation='relu')(input_img)

output_img = Dense(input_size, activation='sigmoid')(code)

autoencoder = Model(input_img, output_img)

autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train, epochs=5)
plot_autoencoder_outputs(autoencoder, 5, (28, 28))
weights = autoencoder.get_weights()[0].T
n = 10
plt.figure(figsize=(20, 5))
for i in range(n):
    ax = plt.subplot(1, n, i + 1)
    plt.imshow(weights[i+20].reshape(28, 28))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
noise_factor = 0.4
x_train_noisy = x_train + noise_factor * np.random.normal(size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(size=x_test.shape)
x_train_noisy = np.clip(x_train_noisy, 0.0, 1.0)
x_test_noisy = np.clip(x_test_noisy, 0.0, 1.0)
n = 5
plt.figure(figsize=(10, 4.5))
for i in range(n):
    # plot original image
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    if i == n/2:
        ax.set_title('Original Images')
    # plot noisy image
    ax = plt.subplot(2, n, i + 1 + n)
```

```
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    if i == n/2:
        ax.set_title('Noisy Input')
input_size = 784
hidden_size = 128
code_size = 32
input_img = Input(shape=(input_size,))
hidden_1 = Dense(hidden_size, activation='relu')(input_img)
code = Dense(code_size, activation='relu')(hidden_1)
hidden_2 = Dense(hidden_size, activation='relu')(code)
output_img = Dense(input_size, activation='sigmoid')(hidden_2)
autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train_noisy, x_train, epochs=10)
n = 5
plt.figure(figsize=(10, 7))
images = autoencoder.predict(x_test_noisy)
for i in range(n):
    # plot original image
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    if i == n/2:
        ax.set_title('Original Images')
    # plot noisy image
    ax = plt.subplot(3, n, i + 1 + n)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
```

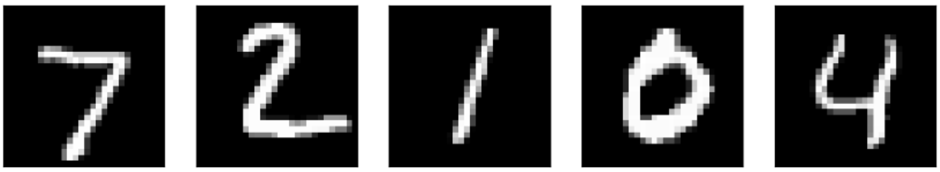
```
    ax.get_yaxis().set_visible(False)
    if i == n/2:
        ax.set_title('Noisy Input')
    # plot noisy image
    ax = plt.subplot(3, n, i + 1 + 2*n)
    plt.imshow(images[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    if i == n/2:
        ax.set_title('Autoencoder Output')
```

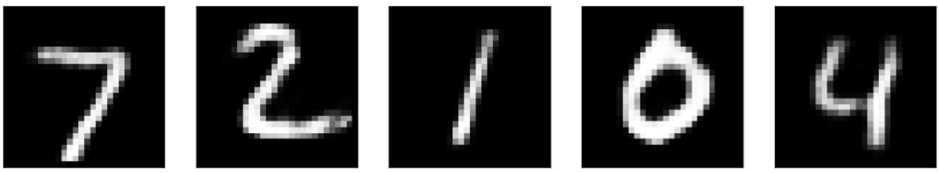
Output:

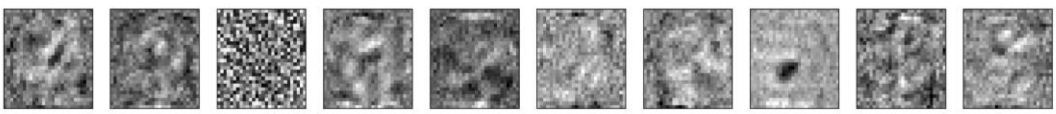
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
(60000, 784)
(10000, 784)
```

```
Epoch 1/3
1875/1875 [==============================] - 11s 5ms/step - loss: 0.1365
Epoch 2/3
1875/1875 [==============================] - 10s 5ms/step - loss: 0.0973
Epoch 3/3
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0915
<keras.src.callbacks.History at 0x78f0b11511b0>
```

```
313/313 [==============================] - 1s 2ms/step
```

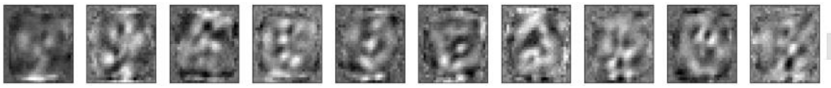






```
Epoch 1/5
1875/1875 [==============================] - 5s 2ms/step - loss: 0.1617
Epoch 2/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1038
Epoch 3/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0964
Epoch 4/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0951
Epoch 5/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0947
<keras.src.callbacks.History at 0x78f092210280>
```

```
313/313 [==============================] - 0s 1ms/step
```



```
Epoch 1/10
1875/1875 [==============================] - 11s 5ms/step - loss: 0.1639
Epoch 2/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1269
Epoch 3/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1197
Epoch 4/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1161
Epoch 5/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1141
Epoch 6/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1127
Epoch 7/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.1116
Epoch 8/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1108
Epoch 9/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1101
Epoch 10/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.1095
<keras.src.callbacks.History at 0x78f0907b6440>
```

```
313/313 [==============================] - 1s 2ms/step
```